



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

1969-06

A universal syntax checker.

Leahy, John Francis III

Monterey, California. Naval Postgraduate School

<http://hdl.handle.net/10945/13338>

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

<http://www.nps.edu/library>

NPS ARCHIVE
1969
LEAHY, J.

A UNIVERSAL SYNTAX CHECKER

by

John Francis Leahy

United States Naval Postgraduate School



THESIS

A UNIVERSAL SYNTAX CHECKER

by

John Francis Leahy III

T 131 863

June 1969

This document has been approved for public release and sale; its distribution is unlimited.

A Universal Syntax Checker

by

John Francis Leahy III
Lieutenant, United States Navy
B.S., United States Naval Academy, 1960

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1969

NPS ARCHIVE ~~Thesis L3578c.1~~

1969

LEAHY, J.

TABLE OF CONTENTS

I.	INTRODUCTION -----	7
II.	DEFINITIONS -----	11
	A. SYNTAX AND SEMANTICS -----	11
	B. GRAMMAR AND LANGUAGE -----	14
	C. RELATED RESEARCH -----	17
	D. PARSING METHOD -----	20
	1. Grammar -----	21
	2. Derivations -----	21
	3. Syntax Trees -----	22
	4. Procedure for a top-down left-right parser -----	23
III.	SUPPORT ROUTINES -----	24
	A. DISCUSSION -----	24
	B. TABLES -----	24
	1. B.N.F. Table (P) -----	24
	2. Symbol Table (SYMTAB) -----	25
	3. Onright Table (ONRIGHT) -----	26
	4. Production Table (PR) -----	27
IV.	THE PARSING ALGORITHM -----	28
	A. BACKGROUND -----	28
	B. PROCEDURES -----	28
	1. Nextsym -----	28
	2. Recognize -----	29
	C. THE SYNTAX CHECKING ALGORITHM -----	29

V. CONCLUSIONS -----	35
A. USES -----	35
B. FURTHER RESEARCH AND IMPROVEMENTS -----	36
APPENDIX A Backus-normal form requirements -----	37
COMPUTER OUTPUT -----	38
COMPUTER PROGRAM -----	54
BIBLIOGRAPHY -----	62
INITIAL DISTRIBUTION LIST -----	64
FORM DD 1473 -----	65

ACKNOWLEDGEMENTS

The author wishes to acknowledge the enthusiastic support of Professor Gary A. Kildall who provided a continuous stream of knowledge, support and spirit.

I. INTRODUCTION

The last decade has seen the general acceptance of the importance of syntax description and syntactical analysis in the development of, and support for, new automatic programming languages for digital computers.

Since the introduction of the stored program computer, communication between man and his computer has been a problem. The laborious and time consuming numeric coding of machine language inhibited many potential computer users from learning how to communicate directly with the computer. The man with a computer solvable problem often found manual calculations easier than trying to communicate with either the expert programmer or the machine. A means of communication with the computer, more easily learned than machine language, was needed. This led to the development of problem-oriented languages.

Problem-oriented languages like ALGOL, PL/1, and FORTRAN, were designed to facilitate communication between user and computer for solutions to mathematical, and other special interest problems.

These automatic programming languages generally consisted of a vocabulary which incorporated many key words from a profession or an interest area. The user could instruct the computer in a language similar to ordinary English usage. For example, instead of trying to manipulate an array by laboriously coding a sequence of instructions to achieve the transpose of a matrix, the user could achieve the same goal by the use of a special reserved word, such as "TRANSPOSE". It should be noted, however, that these special reserved words, such as

"TRANSPOSE", have a specific, or nonredundant meaning. The programmer must use these words in accordance with the coding restrictions of the particular language in which he is programming.

The advent of the automatic programming languages attracted more users to the computer. These new users, as they discovered more applications for the computer, requested more languages specifically designed for their particular interests. Computer specialists responded by developing more new languages.

The introduction of time sharing systems, with many terminals, magnified the problem of satisfying many users and their programming language requirements. Whereas previously only the large corporations could afford a computer installation, now many small businesses were able to share a central installation. The over-all effect provided a large body of users, solving computer problems with a wide variety of special purpose languages.

The cost of these time sharing systems is distributed among the users. Each user is charged for the amount of computer processor time used by his program. Obviously, the user is interested in minimizing his processor time, thus decreasing costs and increasing profits. However, many of the users of time sharing systems are not completely familiar with the restrictions of the particular language in use. As a result, they use the computer to "trouble shoot" their programs for syntax errors. An experienced programmer finds that programs seldom compile correctly on the first attempt. Yet compilation of entire programs is attempted on each occasion with the associated consumption of expensive processor time. The need is apparent for a tool that will assist the programmer in eliminating syntax errors, while minimizing expensive processor time.

A by-product of reducing the number of attempts to compile is the additional processor time available. This time can be used to either improve response time to the current users, or to provide service to new users.

Thus the objective of this thesis: to provide a basic universal syntax checking program which could be expanded for use in a time-sharing environment. This program is hereafter referred to as the Universal Syntax Checker. This syntax checking program accepts the description of all languages and provides a syntax check of programs written in the described languages, as shown in Figure 1-1. The syntax checker requires less facility resources than a language processor, generates no code, and is intended for use in a time-sharing environment where the various users may time-share the universal syntax checker regardless of the language in use at each terminal.

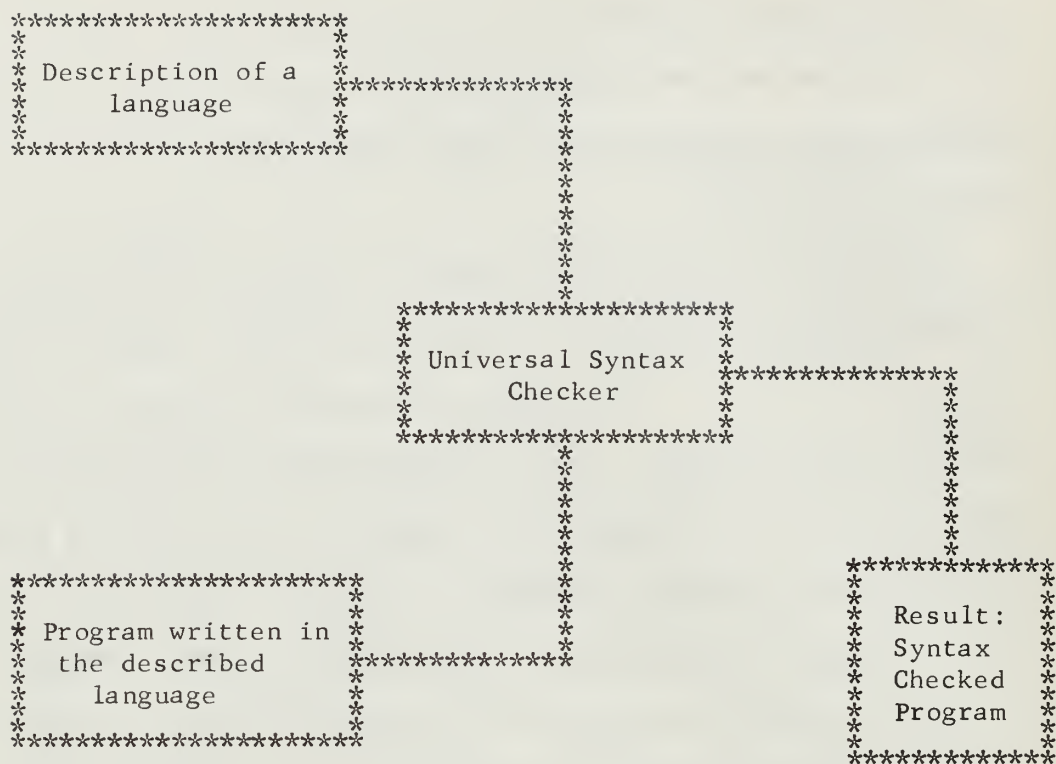


Figure 1-1 Universal Syntax Checker

II. DEFINITIONS

A. SYNTAX AND SEMANTICS

In order to understand the operation of the universal syntax checker, a knowledge of the methods used to formulate and describe languages is necessary.

There is a lack of standard uniform notation throughout the literature, therefore the definitions and notation from [1] and [2] are selected for use in this discussion.

The problem of describing a language involves a unique difficulty. During the discussion, a distinction must be made between the language being described and the describing language. If the language being described is called the "language", then the language in terms of which the description is being made is called the "metalanguage". If these two languages are not distinguishable, much ambiguity and imprecision results. A discussion of how languages are described is in order.

In the exposition of automatic programming languages, the descriptions themselves can be classified into two types; syntactic definitions and semantic discussions. The syntactic definitions make explicit what structures are to be meaningful in the language. These definitions are concerned with the proper construction of words, expressions, and statements. The semantic discussions are concerned with the meanings to be attributed to the various structures and their proper usage in the language.

It often happens that in the process of describing a new computer language, the formats for statements are presented in terms of the natural language. These formats define which components are necessary and how they are to be put together to form meaningful statements. These are the syntax and semantics of the language.

The syntax of a FORTRAN DO statement can be given as the natural language definition:

DO n i = m_1 , m_2 , m_3 , where,

n is a statement identifier,

i is a simple integer,

and m_1 , m_2 , m_3 are simple integer variables.

The syntax definition, however, is not sufficient to specify a properly formed and meaningful DO statement. It is also necessary to discuss the semantics of the DO statement. The semantics could be given as follows:

$$V(m_2) > V(m_1) > 0 \text{ and } V(m_3) > 0$$

where $V(m_i)$ is the value at execution of the variable or constant m_i . It is also necessary that n refer to a statement not previously defined in the subprogram [1].

In general, the syntax and semantics of an entire language, (i.e., of each statement in the language), are given in order to specify the proper construction of meaningful statements in that language.

To formalize the definitions in the metalanguage, each definition is given the form of a statement or construct, sometimes called a production [3]. Thus the syntactic definition of an "unsigned integer" could be:

An unsigned integer can be formed by writing
a digit or
an unsigned integer followed by a digit.

This definition is also an example of a recursive definition. That is, the term "unsigned integer" is defined in terms of itself.

Syntactic definitions can frequently be shortened by using a metalanguage symbolism. The following symbols will be used:

<u>SYMBOL</u>	<u>INTERPRETATION</u>
< x >	angular brackets; a syntactic category, the object named x.
::=	"can be written as" or, "is defined as".
	reads as "or".

To repeat the recursive definition of an unsigned integer using the above notation:

$$\begin{aligned} < \text{unsigned integer} > ::= < \text{digit} > | \\ & \qquad \qquad \qquad < \text{unsigned integer} > < \text{digit} > \end{aligned}$$

This method of syntactic definition of a language is called "specification by Backus-normal form," and abbreviated B.N.F.¹.

The formalism for the semantic discussion of a particular language is not readily available or apparent. Although attempts have been made to make this formalization [1], it is not of concern here since the universal syntax checker verifies the syntax, not the semantics of a particular language.

¹Actually, the form is not normal, B.N.F. is often called Backus-Naur form. Naur introduced the actual notation, and Backus introduced the concept [2].

B. GRAMMAR AND LANGUAGE

As mentioned earlier, the universal syntax checker accepts the definition of a language and programs written in the language.

The second input to the checker, (a program written in the described language), necessitates a discussion of grammar.

Mention of the word "grammar" brings to mind many thoughts. Most people relate the word to something learned in school, or think of a set of words and rules which describe some language. Webster defines grammar in the following manner:

The science treating of the classes of words, their inflections, and their syntactical relations.

Most of us recall dissecting and diagramming sentences to learn the syntactical relation of the parts of a sentence. This is known as parsing. For example, the sentence "The little girl talks fast.", may be dissected after a syntactic definition of a small subset of English.

1	< sentence >	::=	< noun phrase >	< verb phrase >
2	< noun phrase >	::=	< adjective >	< noun phrase >
3			< adjective >	< singular noun >
4	< verb phrase >	::=	< singular verb >	< adverb >
5	< adjective >	::=	the	
6			little	
7	< singular noun >	::=	girl	
8	< singular verb >	::=	talks	
9	< adverb >	::=	fast	

Figure 2-1. Constructs of a sentence

Through the application of the syntax rules the sentence is parsed. The results, when diagrammed, yield a syntax tree. Figure 2-2 shows the diagram of this particular syntax tree.

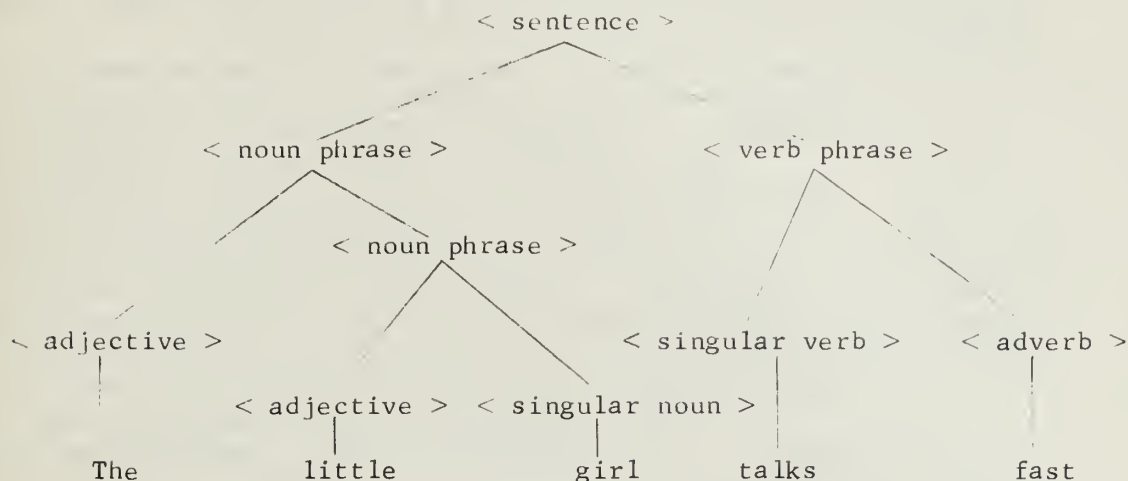


Figure 2-2. Diagram of the sentence
"The little girl talks fast."

The application of the rules in Figure 2-1 to the sentence "The little girl talks fast." reveals that the sentence is grammatically correct. One should observe that sentences cannot only be tested but may also be generated by the application of the same rules. Thus application of rule 1 followed by as many applications of rule 2 as desired, and so on until no further application of rules is possible, would also yield a grammatically correct sentence. The sentence might not mean anything, but it would be grammatically correct. The semantics of a language prevent improper statement formulation. For example, the application of rule 1, followed by the application of rule 2 twice, would yield the sentence, "The the little girl talks fast," which is grammatically correct, but nonsensical just the same.

To formalize the grammar presented, four ingredients were necessary. First, the ingredient of syntactic categories, the noun phrase, adjective,

etc., from which strings of words were derived. These syntactic categories are called non-terminal symbols. Second, there were the words from which the sentences could be constructed. These objects are called terminal symbols. Third, the relations between the various strings of terminal and non-terminal symbols which are called productions. These productions are the rules of syntax, as shown in Figure 2-1. And last, there was one distinguished non-terminal symbol, which appeared nowhere on the right of some production rule. This distinguished symbol is referred to as the start symbol [4], or the goal symbol [2]. The non-terminal symbol "sentence" was the goal symbol in the example given.

Thus a grammar G may be defined as (V_n, V_t, P, Z) . The symbols V_n, V_t, P , and Z represent in order, the set of non-terminal symbols, the set of terminal symbols, a set of productions, and the goal symbol.

Let V_n^*, V_t^*, V^* be finite concatenations of symbols from V_n, V_t , and V_n union V_t respectively. These concatenations are called strings.

Let $:\stackrel{*}{\underset{G}{=}}$ represent the application of a finite sequence of productions from G . $\beta \in V_t^*$ is said to be a "derivation" of $\alpha \in V_n$ if and only if there exists a sequence of productions such that $\alpha :\stackrel{*}{\underset{G}{=}} \beta$.

Let G be the subset of English defined earlier. Letting $\alpha = \langle \text{sentence} \rangle$, $\beta = \text{"The little girl talks fast."}$, the parsing tree of Figure 2-2 shows that $\langle \text{sentence} \rangle :\stackrel{*}{\underset{G}{=}} \text{"The little girls talks fast."}$

A language $L(G)$ is defined as $\{w | w \text{ is in } V_t^* \text{ and } Z :\stackrel{*}{\underset{G}{=}} w\}$ [4]. That is, a string is in $L(G)$ if and only if:

1. The string consists of a finite string of terminal symbols,
and
2. the string can be derived from Z by the application of a
finite sequence of productions from the grammar G .

The set of finite strings of terminal symbols defined by a grammar G is called the set of final sentential forms.

Now the additional input to the universal syntax checker may be firmly identified. The statement, "and a program written in the described language," means that the syntax checker is provided a string (program) which is a member of V_t^* .

Thus to summarize, the universal syntax checker receives the following two inputs:

1. A description of a language L in B.N.F. (The productions, non-terminal symbols, and terminal symbols.)
2. A finite string of terminal symbols from the described language. (An element of V_t^* .)

A formal description of the universal syntax checker is now possible.

Given the B.N.F. of a language L , and a finite string of terminal symbols from the described language, V_t^* , the universal syntax checker will determine whether that string is, or is not, a member of $L(G)$ for the language L defined.

C. RELATED RESEARCH

The use of "syntax-directed" techniques is not a new one. This technique has been used in constructing natural language translators [5], compilers [6,7], automatic programming language translators [8, 9, 10], and context-free grammar recognizers [11].

Syntax directed analysis of natural languages is an unsolved problem due primarily to the inability to precisely define the syntax of each language. An excellent survey of the techniques employed can be found in Bobrow [5].

Griffiths and Petrick, [12] describe many types of recognition procedures, all syntax-directed, for context-free grammars. They employ Turing Machine algorithms to highlight the various methods. Turing machines were used to preserve clarity, conciseness, and to allow comparisons of procedures on the same level of complexity. The universal syntax checker more closely approaches the selective top to bottom (STB) method than any other described.

Irons [6], develops a syntax-directed ALGOL 60 compiler. The arrangement of various tables to contain the specifications of the language are very similar to those employed in the universal syntax checker. Each specifies entries for all syntactic units, as integers, in the described language with a one-to-one correspondence between the actual symbol vector and the integer representation of that description. The two methods do differ in that Irons uses a bottom-to-top method. Thus, his need for a precedence matrix to ensure that the longest string possible is accepted. The syntax checker requires reordering of the B.N.F. to present the longest string possible before any shorter string.

Feldman and Gries discuss the pushdown stack method [3], as one of the two ways to achieve a parsing process. This same method is an integral part of the "cellar principle" used to design a syntax controlled generator of formal language processors [13]. The cellar principle is based on sequential processing of the input symbols.

The syntax checker also employs a sequential processing of symbols and the recursive procedures available in PL/I incorporate the pushdown stack implicitly for all variables.

Barnett and Futrelle presents an account of the SHADOW language that is used to describe a syntax of a language and an associated subroutine which parses an input string [14]. The method requires that a mnemonic argument, in addition to the input string, be provided to the SHADOW subroutine. These arguments include the names of arrays which contain the string and the syntax. Thus, if a parse of a rational fraction is desired the mnemonic RATFRN is a required input argument. The appearance of this mnemonic, requesting a syntactic analysis, causes the subroutine to use the most recently read requested pattern name and input string. This system is obviously unsuited for the time-sharing environment due to the requirement that the programmer be familiar with the syntax of the language in use.

Unger presents a Global Parser (GP), for phrase structured grammars [11]. The method he employs is also very close to the method used in the universal syntax checker. Some major differences do exist. The primary difference is his use of a set of routines for determining possible prefixes and suffixes of N-derivable strings, finding the minimum length of such strings, and sub-strings that can never appear in N-derivable strings. Another difference is the method of comparison between the production and input string. Unger claims that by matching all the terminal symbols of the intermediate string against the input string and constantly partitioning the input string, a broad class of checks can be made to terminate fruitless paths quickly. This parser will not handle cyclic definitions.

An error correcting parse algorithm is described by Irons utilizing a syntax-directed scheme [15]. The algorithm provides two services. First, the algorithm provides a parse of strings written in the language described. Second, if an incorrect string is presented, the algorithm will make substitutions, insertions, and deletions to make the object string syntactically correct. The approach is different from the syntax checker in that all possible parses are carried along until one can be determined to be correct. Backtracking and recursive productions are not allowed. Recursive productions are replaced by a similar powerful definition, which allows iterations in pairs of symbols.

Most authors agree that there are several advantages and disadvantages to syntax-directed procedures. Among the advantages are the simplicity of compiler construction and the ability to change the specifications of a particular language. In addition languages may be switched by merely changing the contents of the syntax table. Further, the syntax-directed parser can take into account as large a context as is required to perform the parsing. The disadvantages include the difficulty in trying to specify the syntax of a language, the fact that syntax-directed compilers contribute little toward the generation of optimum code, and generally poor error analysis and recovery. Error type determination is nearly impossible.

D. PARSING METHOD

It has been shown that taking a string of symbols and a grammar, and constructing a derivation of the string to form its syntax tree, is called parsing, recognizing, or analyzing.

There are two basic types of parsing methods: top-down and bottom-up. The bottom-up method will not be covered but is explained in [3]. The top-down parser gets its name from the fact that it is goal oriented. The top-down parser starts with the most global production (goal symbol) and works its way down the productions, attempting to match the input string. Each method can be further qualified as left-right or right-left, depending on the order of processing the symbols.

In order to discuss the manner employed by the syntax checker to accomplish its assigned tasks, consider the following example of a language and a method to parse such a language:

1. Grammar

Given the following grammar;

$$\begin{aligned} \langle Z \rangle &::= \perp \langle a \rangle \perp \\ \langle a \rangle &::= \langle b \rangle \mid \\ &\quad \langle a \rangle + \langle b \rangle \\ \langle b \rangle &::= \langle c \rangle \mid \\ &\quad \langle b \rangle - \langle c \rangle \\ \langle c \rangle &::= T \\ &\quad (\langle a \rangle) \end{aligned}$$

Non-terminal symbols: Z, a, b, c .

Terminal symbols: $T, \perp, +, -, (,)$.

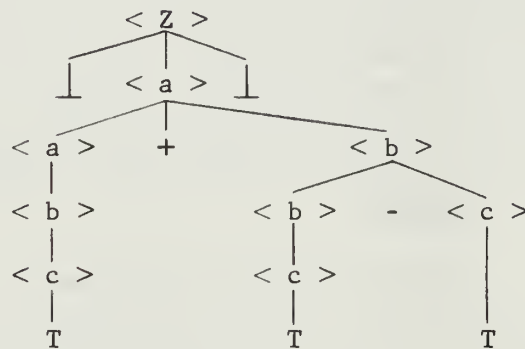
2. Derivations

To discuss ambiguity it is necessary first to discuss derivations. Observe that the following string has more than one derivation. Thus, $\langle c \rangle + \langle b \rangle - \langle c \rangle$ can be derived by two sets of productions:

$\langle Z \rangle ::= \perp \langle a \rangle \perp ::= \langle a \rangle + \langle b \rangle ::= \langle b \rangle + \langle b \rangle$
 $::= \langle c \rangle + \langle b \rangle ::= \langle c \rangle + \langle b \rangle - \langle c \rangle$ and,
 $\langle Z \rangle ::= \perp \langle a \rangle \perp ::= \langle a \rangle + \langle b \rangle ::= \langle a \rangle + \langle b \rangle - \langle c \rangle$
 $::= \langle b \rangle + \langle b \rangle - \langle c \rangle ::= \langle c \rangle + \langle b \rangle - \langle c \rangle$

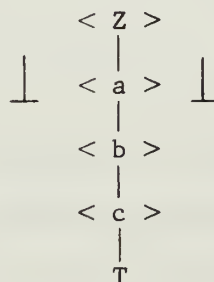
3. Syntax Trees

The construction of the syntax tree is now shown;



Although there may be more than one set of productions which yields the same string of symbols, their syntax-trees are the same. Ambiguity exists only when one or more strings have more than one syntax tree.

Observe that the syntax tree shown below would result if the string $\langle c \rangle + \langle b \rangle - \langle c \rangle$, not a member of $L(G)$, is parsed by proceeding left at every junction down the tree (applying the left-most derivation).



4. Procedure for a top-down left-right parser

To acquire familiarity with the top-down left-right parser, consider the following procedure. The top-down left-right parser makes an initial assumption that the string presented is a valid final sentential form, and then proceeds as follows:

- step 1. Establish goal symbol, in this case Z.
- step 2. Apply a production to the goal or subgoal.
- step 3. First symbol of resultant string a terminal?
Yes, continue; No, go to step 6.
- step 4. Terminal symbol and input string match?
Yes, continue; No, go to step 7.
- step 5. More symbols in string of production?
Yes, continue; No, done.
- step 6. Establish subgoal and go to step 2.
- step 7. An alternate, (OR), production?
Yes, go to step 6; No, done.

A slightly modified form of the above procedure will recognize grammars with left, right, and self-embedded recursion. In addition, ambiguous grammars are processed such that when two or more syntax trees are available, the first, as determined by its placement in the B.N.F., will be recognized.

This simplified procedure has been provided to acquaint the reader with the general procedure of top-down left-right parsing.

III. SUPPORT ROUTINES

A. DISCUSSION

The previous chapters presented the schema of the universal syntax checker. Included was an explanation of the input requirements of the system as well as a simplified method to accomplish the syntax checking task.

Before a detailed discussion of the actual parsing algorithm can begin, it is necessary to describe certain grammar manipulations and tables required to support the algorithm.

B. TABLES

The manner in which the grammar is placed into the supporting tables is now presented. Note that the identifiers enclosed in the parentheses immediately following the table titles, are the actual identifiers which were used in the coding of the universal syntax checker program.

1. B.N.F. Table (P)

This table contains the definitions of the B.N.F. in the form $\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \mid \langle \text{identifier} \rangle \langle \text{letter} \rangle$, and appears in the table as shown in Figure 3-1, where i is the number of symbols in the production and j is the number of productions describing the language. The example is taken from the first grammar listed in the computer program listing.

P	1	2	3	...	i
1	IDENTIFIER	LETTER			
2		IDENTIFIER	LETTER		
3	LETTER	A			
4		B			
5		C			
⋮		⋮			
j					

Figure 3-1. The B.N.F. table.

2. Symbol Table (SYMTAB)

The symbol table is constructed from the grammar placed in the B.N.F. table. This symbol table contains all the elements of V_n followed by all the elements of V_t . The first entry in the table is the empty string.

Since no terminal symbol may appear as the left part of a production, the non-terminal symbols were located by examining the first column of the B.N.F. table and applying the following logic:

step 1. Is the symbol being examined listed in the symbol table?

Yes, examine next symbol; No, go to step 2.

step 2. Put the symbol in the symbol table, increment the table pointer, and examine the next symbol.

After noting the location of the last non-terminal symbol, the terminal symbols are placed into the table. Since no terminal symbol may appear on the left of a production, these symbols are determined using the logic as before, excluding the first column

and examining the remaining columns of the B.N.F. table. Referring to the grammar shown in Figure 3-1, Figure 3-2 shows the symbol table upon completion of its construction. The identifier, nsymbols, refers to the total number of symbols in the grammar.

3. Onright Table (ONRIGHT)

There is a one-to-one correspondence between the subscript numbers of the symbol table and the subscript numbers of the onright table.

SYMTAB

1	IDENTIFIER
2	LETTER
3	
⋮	
40	last non-terminal
41	A
42	B
43	C
⋮	
nsymbols	(last symbol)

Figure 3-2. The Symbol table.

Thus, ONRIGHT (i) is marked "true" if and only if the symbol in SYMTAB (i) appears in the right part of some production. A search of the "false" entries in the onright table produces the goal symbol. More than one "false" entry indicates that the goal symbol is not unique and therefore the grammar is unacceptable. Figure 3-3 depicts the onright table for an acceptable grammar.

ONRIGHT

1	True
2	False
3	True
4	True
:	""
nsymbols	True

Figure 3-3. The Onright table.

4. Production Table (PR)

There is a one-to-one correspondence between the production table (PR) and B.N.F. table (P). Also, there exists an onto mapping from the entries in the production table to the subscript numbers of the symbol table. Thus, wherever the symbol in SYMTAB (i) appears in the B.N.F. table, the entry "i" is made in the production table. For example, consider the symbol table in Figure 3-2. Wherever the symbol IDENTIFIER appears in the B.N.F. table shown in Figure 3-1, an entry "2" is made in the production table as shown in Figure 3-4. Note the recursive production occurring in the second row.

PR

	1	2	3	4	5	...	i
1	2	3 ₁					
2	2	2	3				
3	3	4 ₁					
4	3	4 ₂					
5	3	4 ₃					
:							
j							

Figure 3-4. The Production table.

IV. THE PARSING ALGORITHM

A. BACKGROUND

The inputs, general parsing procedure, and support tables of the universal syntax checker have been described. A presentation of the actual programming implementation remains. This implementation was accomplished utilizing the recursive procedures available in the PL/1 programming language. The operation of the major procedure, called RECOGNIZE, is dependent upon a symbol buffer and accumulator. The buffer contains a portion of the input string. The accumulator provides symbol back-up and is the heart of the procedure NEXTSYM. These two procedures will be discussed briefly before a description of the algorithm is given in reference ALGOL [16].

B. PROCEDURES

1. Nextsym

The NEXTSYM procedure centers around the accumulator shown in Figure 4-1. Each element of the accumulator contains an entry "i" corresponding to SYMTAB (i) for each symbol recognized in the buffer.

Associated with the accumulator are three pointers: ap, tap, and acclen. The accumulator pointer, ap, points to the symbol being examined. The temporary accumulator pointer, tap, retains the value of the accumulator pointer at each junction in the syntax tree that the procedure examines. The accumulator length pointer, acclen, retains the total number of accumulator positions filled.

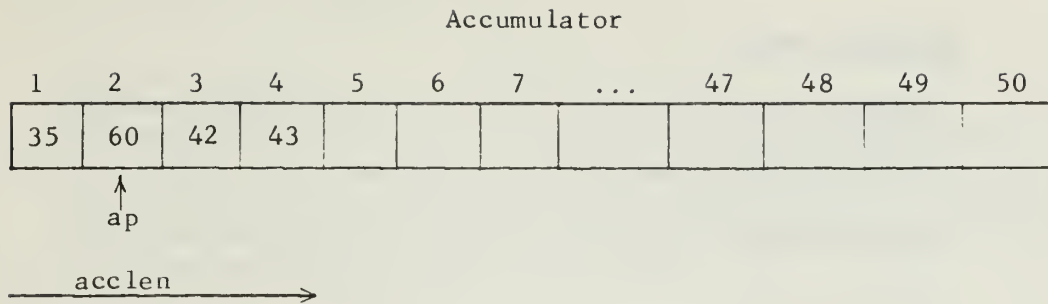


Figure 4-1. Accumulator

The accumulator will hold the first fifty symbols from the input string. Thereafter, it will contain the most recent forty to fifty symbols. The variable "offset" serves to adjust the subscript number *i* of the accumulator while allowing the accumulator pointer to increase sequentially.

2. Recognize

The RECOGNIZE procedure is a top-down left-right slow-back parser. The method, as described in Chapter II, has been implemented with one exception: the parser attempts to recognize the left-most derivation first. The B.N.F. presented to the universal syntax checker must be modified so that the left-most derivation is the longest string possible (see Appendix A).

C. THE SYNTAX CHECKING ALGORITHM

The description of the syntax checking algorithm follows (reference ALGOL). The ALGOL version of the algorithm has not been run on a computer, although the PL/1 version presented in the computer program section has been tested, and the results are in the computer output section.

```

procedure MAIN;

  integer bp, ap, acclen, offset, i, j, k, npr, nonterm,
  nsymbols, bl, linecount, target, scan, a;

  array Symtab [1:nsymbols], Buffer [1:80], Accum [0:50],
  PR [1:n, 1:8];

  boolean array Onright [1:nsymbols];

  boolean change, empty, done, check;

comment This procedure is a top-down left-right slow-back parser.

  To recognize a final sentential form of the B.N.F. presented to
  the procedure, the distinguished symbol is established as the
  goal symbol, and productions are applied, where valid, in the
  order presented. Once a terminal symbol is encountered, the
  symbol is matched with the next symbol in the input string.
  If the match is successful, an attempt is made to match the
  next symbol of the string, and so on. If a match is not
  found, then an alternate production is attempted in an effort
  to recognize the symbol in the buffer. On completion of the
  procedure, the input string will be declared syntactically
  correct or incorrect;

procedure LOOKUP (k);

  value k; integer k;

  begin for j := 1 until npr do

    if PR(j,1) = k then

      LOOKUP := j;

      go to EXIT end;

  EXIT: end LOOKUP;

```

```

boolean procedure NEXTSYM (k);

  value k; integer k;

begin comment This procedure controls the accumulator and checks
  for recognition of the symbols in the buffer with the results
  of the application of production rules;

integer i, m;

i := (ap + 1) - offset;

if i > acclen then begin
  for i := i while (true)
    if i ≤ 50 then begin
      for j := acclen + 1 step 1 until i do begin
        accum [j] := SCAN;

        if check then write ( * symtab [accum[j]]); end;

      acclen := i;

      if accum [i] = k then begin
        ap := ap + 1;

        NEXTSYM := true; end

      else NEXTSYM := false;

      go to A; end
    else begin offset := offset + 10;

      begin for m := 1 until 40 do
        accum [m] := accum [m + 10]; end;

      i := i - 10; end
  else if i < 1 then begin
    write (' depth of search exceeded');

    go to A; end

  else begin

```

```

    if accum [i] = k then begin

        ap := ap + 1;

        NEXTSYM := true; end

    else NEXTSYM := false;

end;

A: end NEXTSYM:

boolean procedure RECOGNIZE (production, element);

    value production, element;

    integer production, element;

begin comment This procedure attempts to recognize each of the
    symbols in the input string. If a symbol is recognized then
    RECOGNIZE is set to true, otherwise false;

integer k, lr, tap;

lr := 0;

if element  $\leq$  8 then tap := ap;

    if element = 1 then begin

        if PR [production, 1] = PR [production, 2] then

            if  $\neg$  RECOGNIZE (production, 3) then begin

                ap := tap;

                RECOGNIZE := if PR [production, 1] = PR [production + 1, 1]

                    then RECOGNIZE (production + 1, 1)

                    else false;

            end

            go to OUT;

        end

    else RECOGNIZE := true

    else if RECOGNIZE (production, 2) then begin

        tap := ap;

```

```

if check then begin
    write (symbol found);
    for production := production while (PR [production, 1] =
        PR [production + 1, 1]  $\wedge$  PR [production + 1, 1]  $\neq$ 
        PR [production + 1, 2])
        production := production + 1
    else end;
if lr  $\neq$  0 then begin
    write (symbol, 'number of left recursive
        production found');
    ap := tap;
    RECOGNIZE := true;
    end
    else RECOGNIZE := false
    go to OUT
else begin
    tap := ap;
    RECOGNIZE := if PR [production, 1] = PR [production
        + 1, 1]  $\wedge$  PR [production + 1, 1]  $\neq$ 
        PR [production + 1, 2] then
        RECOGNIZE (production + 1, 1)
        else false;
    go to OUT;
    end
else begin
    k := PR [production, element];
    RECOGNIZE := if k  $\neq$  1 then

```

```

    if k > nonterm then
        RECOGNIZE := if NEXTSYM (k) then
            RECOGNIZE (production,
                element + 1)
            else false
        else
            RECOGNIZE := if RECOGNIZE (LOOKUP (k), 1)
                then RECOGNIZE (production,
                    element + 1)
                else false
            else true;
    end
else RECOGNIZE := true;
OUT:
end RECOGNIZE;
if  $\neg$ done then
    begin if RECOGNIZE (1, 1) then write ( 'syntax ok' )
        else write ( 'syntax error' ); k := 0;
    for k := k while (k  $\neq$  nsymbols  $\wedge$   $\neg$ done) do k := scan;
    bp := 72; ap := acclen := offset := linecount := 0;
end MAIN;

```

V. CONCLUSIONS

A. USES

The algorithm presented has many applications. However, one application, more suitable than the others, is use in a time-sharing environment.

In a time-sharing system, the universal syntax checker could reside on a direct access storage device, along with the B.N.F. definitions, to be called when desired. Each terminal user would be able to time-share this syntax checker regardless of the language used. The syntax checker would provide a very rapid syntax check for each user at each terminal. As soon as the first syntax error was encountered, syntax checking of that program would end. The user at an on-line terminal would then examine the incorrect statement and effect a correction. The syntax check would restart the program analysis. Figure 5-1 depicts a possible configuration for a time sharing system with syntax checker.

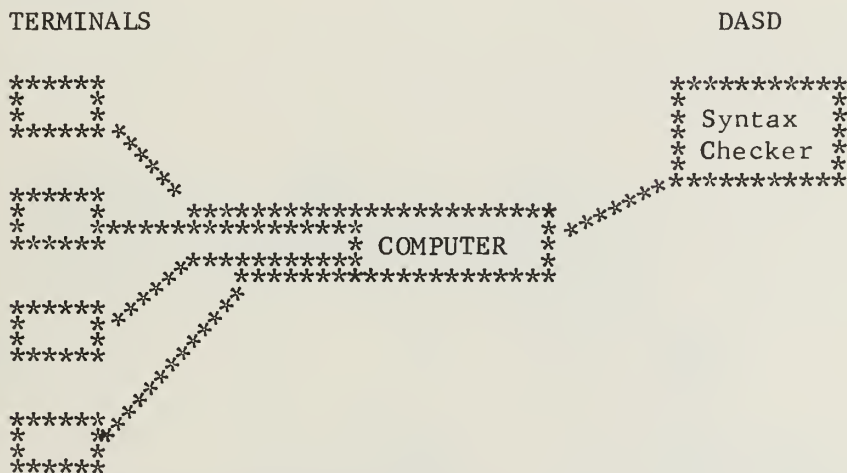


Figure 5-1. Time-sharing system with syntax checker.

B. FURTHER RESEARCH AND IMPROVEMENTS

One obvious need is the actual implementation of the universal syntax checker in a time-sharing system.

Two major improvements are also needed. First, a recovery procedure which will allow recovery to a logical restart point, to continue syntax checking, after encountering a syntax error. Second, there is a need for a more complete and precise set of diagnostic messages. Simply to say "syntax ok" or "syntax error" is not enough. Precise statements of the form where, what, and why are needed.

APPENDIX A

Backus-normal form requirements.

1. Maximum number of characters per symbol is ten.
2. Maximum number of symbols per production is eight.
3. Maximum number of productions per B.N.F. is three hundred.
4. Left recursive productions must include the trivial production prior to the recursive production.
5. Cyclic productions require the trivial productions as in 4.
6. The B.N.F. must be ordered to present the longest string possible prior to any other production.
7. The character string \$PROGRAM is not allowed in a production for a language. This string is used as an indicator to make the end of the B.N.F. and the start of each program submitted for parsing.
8. Place the characters \$PROGRAM immediately after the B.N.F. submitted and immediately after each program submitted for syntax checking.

PRCGRAM	STATEMENT :	END	
STATEMENT	GROUP		
GROUP	PROGRAM		
	MODE		
	CALC	NUMBER	PROGSTATE
	MODE	:	CALCSTATE
PRCGMCDE	STEP	:	
CALC	COMPUTE		
MODE	STEP		
PROGSTATE	PROGSTATE :		PROGSTEP
	PROGSTATE :		
CALCSTATE	CALCSTATE :		CALCSTEP
	CALCSTATE :		
PROGSTEP	COMSTEP		
	INPUT		
	CONDITIONAL		
	CONDSTATE		
CALCSTEP	COMSTEP		
	ARITHEXP		
ARITHEXP	EDITSTATE		
	VARIABLE		
	VARIABLE		
	UNSGNEDUM		
	UNSGNEDUM		
	ARITHEXP		
	ARITHEXP		
COMSTEP	ASSIGNMENT		
	NOTE		
	OUTPUT		
	VARIABLE		
ASSIGNMENT	REPLACEMENT		
VARIABLE	ARITHEXP		
(ARITHEXP		
+			
-			
X			
/			
UNSGNEDUM	NUMBER		
IDENTLIST	UNSGNEDUM		
VARIABLE	VARIABLE		
SUBSCRIPT	IDENTLIST		
IDENT	SUBSCRIPT		
SUBLIST	SIMPLE		
SUBLIST	IDENT		
SIMPLE	ARITHEXP		
IDENT	SUBLIST		
IDENT	IDENT		
LETTER	SIMPLE		
	IDENT		
	LETTER		
	IDENT		
	LETTER		
	A		
	B		
	C		
	1		
	2		
	3		
EDITSTATE	*		
EDITOR	DELETE		
	LINE		
	START		
	READ		
	WRITE		
INPUT	INOUTLIST		
OUTPUT	INOUTLIST		
INOUTLIST	INOUTLIST		
CNC	INOUTLIST		
CONDITIONAL	IF		
	THEN		
	CONDSTATE		

CONDSTATE	TRANSFER	TO	STEP	NUMBER
TRANSFER	GO	START		
ENC	DISCARD	EXECUTE		
SUBOL	TABLE			
NON-TERMINAL				
1			PROGRAM	
2			STATEMENT	
3			GROUP	
4			PROGCODE	
5			CALCSTATE	
6			PROGSTATE	
7			CALCSTATE	
8			PROGSTATE	
9			CALCSTATE	
10			ARITHEXP	
11			COMSTEP	
12			ASSIGNMENT	
13			ADDOF	
14			MULOP	
15			UNSGMEDIUM	
16			IDENTLIST	
17			VARIABLE	
18			SUBSCRIPT	
19			SUBLIST	
20			SIMPLE	
21			IDENT	
22			LETTER	
23			NUMBER	
24			EDITOR	
25			INPUT	
26			OUTPUT	
27			IDENTLIST	
28			CONDITIONAL	
29			CONDSTATE	
30			TRANSFER	
31			END	
32			STEP	
33			COMPUTE	
34			NOTE	
35			(
36			+	
37			-	
38			x	
39			/	
40			A	
41			B	
42			C	
43			1	
44			2	
45			3	
46			*	
47			DELETE	
48			LINE	
49			START	
50				
51				

TERMINAL

[illegible]

```
***** PROGRAM *****  
      COMPUTE : AAA ; EXECUTE  
      *COMPUTE  
      **;  
      *A  
  
FOUNC LETTER  
FOUNC IDENT  
  
FOUNC LETTER #A  
FOUNC LETTER #A  
FOUNC LETTER *:  
  
IDENT --  
FOUNC LETTER  
FOUNC IDENT  
FOUNC LETTER  
FOUNC LETTER  
  
IDENT SIMPLE  
FOUNC VARIABLE  
FOUNC LETTER  
FOUNC IDENT  
FOUNC LETTER  
FOUNC LETTER  
FOUNC LETTER  
  
IDENT --  
FOUNC LETTER  
FOUNC IDENT  
FOUNC LETTER  
FOUNC LETTER  
  
IDENT SIMPLE  
FOUNC VARIABLE  
FOUNC ASSIGNMENT  
FOUNC COMSTEP  
FOUNC CALCSTATE  
FOUNC CALCSTATE  
  
FOUNC #EXECUTE  
FOUNC CALCMODE  
FOUNC GROUP  
FOUNC STATEMENT  
FOUNC END  
FOUNC PROGRAM  
SYNTAX OK  
  
          2 LEFT RECURSIVE PRODUCE  
  
          2 LEFT RECURSIVE PRODUCE  
  
          2 LEFT RECURSIVE PRODUCE  
  
          2 LEFT RECURSIVE PRODUCE  
  
          $PROGRAM  
          2
```

2 \$PROGRAM


```
***** PROGRAM *****  
1  
END  
UNDEFINED SYMBOL END  
SYNTAX ERROR *  
2 $PROGRAM
```

```

***** PROGRAM *****
1  COMPUTE ; AAA ; 888 ; CCCCC ; 121 + 222 ; ( 111 X 222 ) ;
   *COMPUTE
   *;
   *A
FOUND LETTER
FOUND IDENT
FOUND LETTER *A
FOUND LETTER *A
FOUND LETTER *;
IDENT --
FOUND LETTER
FOUND IDENT
FOUND LETTER
FOUND LETTER
IDENT --
FOUND SIMPLE
FOUND VARIABLE
FOUND LETTER
FOUND IDENT
FOUND LETTER
FOUND LETTER
IDENT --
FOUND LETTER
FOUND IDENT
FOUND LETTER
FOUND LETTER
IDENT --
FOUND SIMPLE
FOUND VARIABLE
FOUND ASSIGNMENT
FOUND COMSTEP
FOUND CALCSTEP
FOUND CALCSTATE *B
FOUND LETTER
FOUND IDENT
FOUND LETTER *B
FOUND LETTER *B
FOUND LETTER *;
IDENT --
FOUND LETTER
FOUND IDENT
FOUND LETTER
FOUND LETTER
IDENT --
FOUND SIMPLE
FOUND VARIABLE
FOUND LETTER
FOUND IDENT
FOUND LETTER
FOUND LETTER
IDENT --
FOUND LETTER
FOUND IDENT

```

2 LEFT RECURSIVE PRODUCTIONS FOUND

2 LEFT RECURSIVE PRODUCTIONS FOUND

2 LEFT RECURSIVE PRODUCTIONS FOUND

2 LEFT RECURSIVE PRODUCTIONS FOUND

2 LEFT RECURSIVE PRODUCTIONS FOUND

2 LEFT RECURSIVE PRODUCTIONS FOUND

2 LEFT RECURSIVE PRODUCTIONS FOUND

```

FOUND LETTER
IDENT --
FOUND SIMPLE
FOUND VARIABLE
FOUND ASSIGNMENT
FOUND COMSTEP
FOUND CALCSTEP
FOUND LETTER
FOUND IDENT *C
FOUND LETTER *C
FOUND LETTER *C
FOUND LETTER *C
FOUND LETTER *C
FOUND LETTER *C
IDENT --
FOUND LETTER
FOUND IDENT
FOUND LETTER
FOUND LETTER
FOUND LETTER
FOUND LETTER
FOUND LETTER
IDENT --
FOUND SIMPLE
FOUND VARIABLE
FOUND LETTER
FOUND IDENT
FOUND LETTER
FOUND LETTER
FOUND LETTER
FOUND LETTER
FOUND LETTER
IDENT --
FOUND LETTER
FOUND IDENT
FOUND LETTER
FOUND LETTER
FOUND LETTER
FOUND LETTER
IDENT --
FOUND SIMPLE
FOUND VARIABLE
FOUND ASSIGNMENT
FOUND COMSTEP
FOUND CALCSTEP
FOUND NUMBER *1
FOUND UNSIGNEDUM *2
FOUND NUMBER *1
FOUND NUMBER *1
FOUND NUMBER **
UNSGNEEDUM --
FOUND ADDOP
FOUND ARITHXP
2 LEFT RECURSIVE PRODUCTIONS FOUND

4 LEFT RECURSIVE PRODUCTIONS FOUND

4 LEFT RECURSIVE PRODUCTIONS FOUND

4 LEFT RECURSIVE PRODUCTIONS FOUND

4 LEFT RECURSIVE PRODUCTIONS FOUND

2 LEFT RECURSIVE PRODUCTIONS FOUND

```

```

FOUND CALCSTEP #2
CALCSTATE --
FOUND CALCMODE
FOUND GROUP
FOUND STATEMENT
SYNTAX ERROR

      2 888 ; CCC ; AAA ; EXECUTE

      3 $PROGRAM
      *
SYNTAX ERROR

```

```

3 LEFT RECURSIVE PRODUCTIONS FOUND

```

PARA
SENT
SUBJ
NOUNPHRASE
RELPHRASE
RELPHRASE
ADJSTRING
INTRO
PRED
ART
ADJ
NOUN
REL
COMPARADJ
LOC
IS
EACH


```

***** PROGRAM *****
1  EACH POLYGON SMALLER THAN A TRIANGLE WHICH IS A BLACK SQUARE IS A
  *EACH
    FCLAC ART      *POLYGON
    FCLAC NCUN
    FCUNC NCUNPHRASE
    FCLAC COMPARCJ *SMALLER
    FCLAC INTRC    *THAN
    FCLAC ART      *A
    FCLAC ART      *TRIANGLE
    FCLAC NOUN
    FOUNC NOUNPHRASE
    FOUNC RELPHRASE
    FCUNC RELPHSEST *WHICH
    FCUNC REL      *IS
    FCUNC REL      *A
    FCUNC REL
    FCUNC SUBJ
    FOUNC ART
    FOUNC NOUN
    FCUNC NOUNPHRASE
    FCUNC COMPARCJ
    FCUNC INTRC
    FOUNC ART
    FCUNC NOUN
    FOUNC NOUNPHRASE
    FOUNC RELPHRASE
    FCUNC RELPHSEST
    FCUNC REL
    FOUNC REL
    FOUNC SUBJ
    SYNTAX ERROR
2  HEXAGON PERIOD
3  $PROGRAM

```

```

***** PROGRAM *****
1  EACH POLYGCN SMALLER THAN A BLACK TRIANGLE IS A SQUARE PERIOD COMMA
   EACH
   *POLYGCN
   FOUNC ART
   FOUNC NOUN
   FOUNC NCUNPHRASE
   *SMALLER
   FOUNC COMPARCJ
   *THAN
   FCLNC INTRC
   *A
   FOUNC ART
   *BLACK
   FOUNC ART
   FOUNC ADJ
   FOUNC ACJSTRING
   *TRIANGLE
   FOUNC NCUN
   FOUNC NCUNPHRASE
   FOUNC NCUNPHRASE
   FOUNC RELPHRASE
   FOUNC RELPHSEST
   *IS
   FCLNC SUBJ
   *A
   FOUNC ART
   *SQUARE
   FOUNC NCUN
   FOUNC NCUNPHRASE
   FOUNC PREC
   *PERIOD
   FOUNC SENT
   *COMMA
2  EACH BLACK POLYGCN SMALLER THAN A TRIANGLE IS A SQUARE PERIOD
   EACH
   *BLACK
   FOUNC ART
   FOUNC ART
   FOUNC ADJ
   FOUNC ACJSTRING
   *POLYGCN
   FOUNC NOUN
   FOUNC NCUNPHRASE
   *SMALLER
   FOUNC COMPARCJ
   *THAN
   FCLNC INTRC
   *A
   FOUNC ART
   *TRIANGLE
   FOUNC NOUN
   FOUNC NCUNPHRASE
   FOUNC RELPHRASE
   FOUNC RELPHSEST
   *IS
   FOUNC SUBJ
   *A
   FOUNC ART

```

FGLAC	NGUN	*SQUARE
FOUND	NOUNPHRASE	
FOUND	PREC	*PERIOD
FGLNC	SENT	
FGLNC	PARA	
SYNTAX	CK	
	3	\$PROGRAM

```

***** PROGRAM *****
EACH WHITE POLYGCN GREATER THAN A HEXAGON IS A RECTANGLE PERIOD
*EACH
  *WHITE
    FOUNC ART
    FOUNC ART
    FOUNC ADJ
    FOUNC ADJSTRING
    *POLYGCN
    FOUNC NOUN
    FOUNC NOUNPHRASE
    *GREATER
    FOUNC COMPARACJ
    *THAN
    FOUNC INTRO
    *A
    FOUNC ART
    *HEXAGON
    FOUNC NOUN
    FOUNC NOUNPHRASE
    FOUNC RELPHRASE
    FOUNC RELPHRASE
    *IS
    FOUNC SUBJ
    *A
    FOUNC ART
    *RECTANGLE
    FOUNC NOUN
    FOUNC NOUNPHRASE
    FOUNC PREC
    *PERIOD
    FOUNC SENT

2 $PROGRAM
  *$PROGRAM
    FOUNC ART
    FOUNC ART
    FOUNC ADJ
    FOUNC ADJSTRING
    FOUNC NOUN
    FOUNC NOUNPHRASE
    FOUNC COMPARACJ
    FOUNC INTRO
    FOUNC ART
    FOUNC NOUN
    FOUNC NOUNPHRASE
    FOUNC RELPHRASE
    FOUNC RELPHRASE
    FOUNC RELPHRASE
    FOUNC SUBJ
    FOUNC ART
    FOUNC NOUN
    FOUNC NOUNPHRASE
    FOUNC PREC
    FOUNC PREC
    FOUNC PARA
    FOUNC PARA
    SYNTAX OK

```



```

OFFSET=OFFSET+10;
DO M=1 TO 40;
  ACCUM(M)=ACCUM(M+10);
END;
ACCLEN = 40;
I=I-10;
END;
END NEXTSYM;
SCAN: PROC FIXED BIN;
  DCL T CHAR(10); K CHAR(1); (I,Q,S,N,M) FIXED BIN INITIAL (0);
  ON ENDFILE (SYSIN) GO TO RET;
  IF 'O'B THEN RET;
  DO: DONE='1'B;
    IF S=0 THEN RETURN(N);
  ELSE
    IF N=0 THEN
      DO: PUT LIST('UNDEFINED SYMBOL '||T)SKIP;
      BP=BL; RETURN(0);
    END;
  ELSE
    DO: BP=BP-Q+M-1; RETURN(N);
  END;
END;
DO WHILE ('1'B): BP=BP+1;
  IF BP > BL THEN
    DO: GET EDIT(BUFFER)(A(80)): BP=1;
    LINECOUNT=LINECOUNT+1;
    IF LINECOUNT=1 THEN
      PUT EDIT('***** PROGRAM *****');
    ( X(13), A(29)) PAGE;
    PUT LIST(LINECOUNT);
    PUT LIST('||SUBSTR(BUFFER,BL+1,80-BL))SKIP(2);
  END;
  K=SUBSTR(BUFFER,BP,1);
  IF K='1' THEN
    IF S ^= 0 THEN
      IF N=0 THEN
        DO: PUT LIST ('UNDEFINED SYMBOL '||T)SKIP;
        RETURN(0);
      END;
    ELSE
      DO: BP=BP-Q+M-1;
      RETURN(N);
    END;
  ELSE:
    /* CHARACTER FOUND, SEARCH TABLES */
    DO: S=S+1;
    IF S<=10 THEN

```

```

DO: SUBSTR(T,S,1)=K; K=SUBSTR(T,1,1);
DO I=NONTERM+1 TO NSYMBOLS; THEN /* MAY MATCH */
IF SUBSTR(SYMTAB(I),1,1)=K THEN /* MAY MATCH */
IF SUBSTR(SYMTAB(I),1,S)=SUBSTR(T,1,S) THEN
IF S=10 THEN RETURN(I);
ELSE
IF SUBSTR(SYMTAB(I),S+1,1)=' ' THEN
DO: M=C; N=I; I=NSYMBOLS; Q=S;
END;
ELSE Q=S;
END;
IF N=0 THEN
IF Q<S THEN
DO: AP=BP-Q+M-1;
RETURN(N);
END;
END;
ELSE N,M=0;
END;
END SCAN;
REC: PROC (PROD,ELEM) BIT(1) RECURSIVE;
DCL (PROD,ELEM,K,LR,INITIAL(0),TAP) FIXED BIN;
LOOKUP ENTRY( FIXED BIN) RETURNS(FIXED BIN);
LOOKUP: PROC(I) FIXED BIN;
DCL (I,J) FIXED BIN;
DO J=1 TO NPR;
IF PR(J,1)=I THEN RETURN(J);
END;
END LOOKUP;
IF ELEM > 8 THEN RETURN('1'B);
TAP=AP;
IF ELEM=1 THEN /* MAY BE RECURSIVE SCAN OF PRODUCTIONS */
IF PR(PROD,1)=PR(PROD,2) THEN /* IN RECURSIVE SCAN */
IF ~REC(PROD,3) THEN /* LOOK AT NEXT PRODUCTION */
DO: AP=TAP;
IF PR(PROD,1)=PR(PROD+1,1) THEN
RETURN(REC(PROD+1,1)); ELSE RETURN('0'B);
END;
ELSE /* RECURSIVE PRODUCTION FOUND */
RETURN('1'B);
ELSE /* NOT A RECURSIVE SCAN */
IF REC((PROD),2) THEN /* FOUND PRODUCTION, LOOK FOR
LEFT RECURSION */
DO: TAP=AP; /* FIND FIRST RECURSIVE PRODUCTION */
IF CHECK THEN
PUT LIST(FOUND,1) SYMTAB(PR(PROD,1)) SKIP;
DO WHILE((PR(PROD,1)=PR(PROD+1,1))) {
(PR(PROD+1,1)~=PR(PROD+1,2))}; PROD=PROD+1;

```



```

END:
IF (PR(PROD,1)=PP(PROD+1,1))&
(PR(PROD+1,1)=PR(PROD+1,2)) THEN
DO WHILE(REC(PROD+1,1)); TAP=AP: /* FOUND */
IF CHECK THEN LR=LR+1:
END:
IF LR=0 THEN
PUT LIST(' ',SYMTAB(PR(PROD,1))1'--'11LR11
' LEFT RECURSIVE PRODUCTIONS FOUND')SKIP:
AP=TAP: RETURN('1'B):
END:
ELSE /* PRODUCTION NOT FOUND, LOOK AT NEXT PRODUCTION */
DO: AP=TAP:
IF (PR(PROD,1)=PR(PROD+1,1))&
(PP(PROD+1,1)=PR(PROD+1,2)) THEN
/* NOT A LEFT RECURSIVE DEF SO LOOK AT IT */
RETURN(REC(PROD+1,1)):
ELSE RETURN('0'B):
END:
/* OTHERWISE ELEM IS NOT 1, CHECK NEXT ITEM IN PRODUCTION */
K=PR(PROD,ELEM):
IF K=1 THEN RETURN('1'B):
IF K > NTERM THEN
IF NEXTSYM(K) THEN
RETURN(REC(PROD,ELEM+1)):
ELSE RETURN('0'B):
ELSE
IF REC(LOOKUP(K),1) THEN
RETURN(REC(PROD,ELEM+1)):
ELSE RETURN('0'B):
END REC:
END:
NPR=0:
DO WHILE ('1'B):
GET EDIT ((P(NPR+1,1) DO I=1 TO 8)) (8)A(10)):
IF (D(NPR+1,1)= '$PROGRAM ') THEN GO TO EOF:
NPR = NPR + 1:
PUT EDIT ((P(NPR,1) DO I=1 TO 8)) (8)A(10)) SKIP:
END:
EOF:
NSYMBOLS = 1:
SYMTAB(1) = REPEAT(' ',10): /* SET TOP OF SYMBOL TABLE BLNK*/
DO I = 1 TO 8:
DO J = 1 TO NPR:
DO K = 1 TO NSYMBOLS:
IF P(J,I) = SYMTAB(K) THEN GO TO FOUND:
END:
K,NSYMBOLS = NSYMBOLS + 1:
SYMTAB(NSYMBOLS) = P(J,I):

```

```

FOUND:
  PR(J,I) = K:
  IF I = 1 THEN SUBSTR(ONRIGHT,K,1)='1'R:
  END:
  IF I = 1 THEN NONTERM = NSYMBOLS:
  END:
  NSYMBOLS=NSYMBOLS + 1: SYMTAB(NSYMBOLS) = P(NPR+1,1):
  DO J=2 TO NPR:
    IF PR(J,1) = 1 THEN PR(J,1) = PR(J-1,1):
  END:
  PUT LIST ('SYMBOL TABLE') SKIP:
  PUT LIST ('NON-TERMINAL') SKIP:
  DO I=1 TO NSYMBOLS:
    IF I = NONTERM + 1 THEN
      PUT LIST ('TERMINAL') SKIP:
      PUT LIST (I, SYMTAB(I)) SKIP:
    END:
    PUT LIST ('TRANSLATED PRODUCTIONS') SKIP:
    DO I=1 TO NPR:
      PUT EDIT ((PR(I,J) DO J=1 TO 8))((8)F(8)) SKIP:
    END:
    PUT LIST ('ONRIGHT TABLE') SKIP:
    PUT EDIT ((SUBSTR(ONRIGHT,K,1) DO K=1 TO NSYMBOLS))(R(1)) SKIP:
  J=0:
  DO I=1 TO NONTERM:
    IF (SUBSTR(ONRIGHT,I,1)='0'B) THEN
      DO: J=J+1:
        PUT LIST ('THE UNIQUE TARGET SYMBOL IS ',SYMTAB(I)) SKIP:
        TARGET = I:
      END:
    IF J=1 THEN
      END:
    END:
  END:
  IF J=1 THEN
    DO:
      FAILURE = 1:
      PUT LIST ('THERE IS NO UNIQUE GOAL SYMBOL') SKIP:
    END:
  /*
  /* SET UP THE TEST ARRAY
  /*
  /*
  DO I=1 TO NPR:
    DO J=1 TO 8:
      IF (PR(I,J)>NONTERM) THEN
        TEST(I,J) = 1:
      ELSE TEST(I,J) = PR(I,J):
    END:
  END:
  /*

```

```

/* BEGIN PRODUCTION TESTING */
/*
DO WHILE (CHANGE):
  CHANGE:=0'B:
  DO I=1 TO NPR:
    Z=TEST(I,1):
    IF Z=1 THEN
      DO: EMPTY =1'B:
        DO J=2 TO 8:
          EMPTY = EMPTY & TEST(I,J) =1:
        END:
      IF (EMPTY) THEN
        DO K=1 TO NPR:
          DO J=1 TO 8:
            IF (TEST(K,J)=Z) THEN TEST(K,J) =1:
          END:
        CHANGE = CHANGE | EMPTY:
      END:
    END:
  /* CHECK FOR NON TERMINATING PRODUCTIONS */
  /*
  DO I=1 TO NPR:
    IF (TEST(I,1) =1) THEN
      DO:
        PUT LIST ( 'PRODUCTION ',I,' LEADS TO A NON-TERMINATING
        PHRASE') SKIP:
        FAILURE=1:
      END:
    END:
  /* CHECK FOR EMPTY LEFT AND RIGHT PARTS */
  /*
  DO I=1 TO NPR:
    IF (PR(I,1) =1) THEN
      DO:
        PUT LIST ( ' PRODUCTION ',I,' HAS AN EMPTY LEFT PART') SKIP:
        FAILURE=1:
      END:
    IF PR(I,2) =1 THEN
      DO:
        PUT LIST ( ' PRODUCTION ',I,' HAS AN EMPTY RIGHT PART') SKIP:
        FAILURE=1:
      END:
    /* CHECK FOR IDENTICAL RIGHT PARTS */

```

[illegible]

ASSIGNMENT	OUTPUT	REPLACEMENT	ARITHEXP
VARIABLE	VARIABLE		
(ARITHEXP)
ADD	+		
MUL	*		
UNSIGNED	NUMBER		
IDENTLIST	IDENTLIST		VARIABLE
VARIABLE	SUBSCRIPT		
SUBSCRIPT	IDENT		
SUBLIST	ARITHEXP		
SIMPLE	SUBLIST		
IDENT	SIMPLE		
LETTER	IDENT		
	LETTER		
NUMBER	NUMBER		
EDITSTATE	EDITOR		NUMBER
EDITOR	DELETE		
	LINE		
INPUT	START		
OUTPUT	READ		
IDENTLIST	WRITE		
CONDITIONAL	(
CONDSTATE	IDENTLIST		
	IDENTLIST		
	BOOLEAN		
TRANSFER	TO		
FND	STEP		
	NUMBER		
	CONDSTATE		
	THEN		
	CONDSTATE		

BIBLIOGRAPHY

1. Lee, J. A. N., The Anatomy of a Compiler, p. 1-35, Reinhold, 1967.
2. Ledley, R. S., FORTTRAN IV Programming, p. 32-45, McGraw-Hill, 1966.
3. Feldman, J., Gries, D., "Translator Writing Systems," Communications of the ACM, v. 11 No. 2, p. 77-113, February 1968.
4. Harrison, M. A., Introduction to Switching and Automata Theory, p. 368-391, McGraw-Hill, 1965.
5. Bobrow, D. G., "Syntactic Analysis of English by Computer: A Survey," Proceedings of the Fall Joint Computer Conference, p. 365-387, 1963.
6. Irons, E. T., "A Syntax-Directed Compiler for ALGOL 60," Communications of the ACM, v. 4, No. 1, p. 51-55, January 1961.
7. Cheatham, T. E. Jr., Sattley, K., "Syntax-Directed Compiling," Rosen, S., Programming Systems and Languages, p. 264-297, McGraw-Hill, 1967.
8. Ledley, R. S., Wilson, J. B., "Automatic-Programming Language Translation Through Syntactical Analysis," Communications of the ACM, v. 5, No. 3, p. 145-155, March 1962.
9. Brooker, R. A., and Morris, D. A., "Translation Program for Phrase Structure Languages," Journal of the ACM, v. 9, p. 1-10, 1962.
10. Kuno, S., "The Predictive Analyzer and a Path Elimination Technique," Communications of the ACM, v. 8, No. 7, p. 453-462, July 1965.
11. Unger, S. H., "A Global Parser for Context-Free Phrase Structure Grammars," Communications of the ACM, v. 11, No. 4, p. 240-247, April 1968.
12. Griffiths, T. V., Petrick, S. R., "On the Relative Efficiencies of Context-Free Grammar Recognizers," Communications of the ACM, v. 8, No. 5, p. 289-300, May 1965.
13. Eickel, M., and others, "A Syntax Controlled Generator of Formal Language Processors," Communications of the ACM, v. 6, No. 8, p. 451-455, August 1963.
14. Barnett, M. P., Futrelle, R. P., "Syntactic Analysis by Digital Computer," Communications of the ACM, v. 5, No. 10, p. 515-526, October 1962.

15. Irons, E. T., "An Error Correcting Parse Algorithm," Communications of the ACM, v. 6, No. 11, p. 669-673, November 1963.
16. Naur, P., and others, "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, v. 6, No. 1, p. 1-7, January 1963.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Documentation Center Cameron Station Alexandria, Virginia 22314	20
2. Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3. Chief of Naval Operations (OP-91) Department of the Navy Washington, D. C. 20350	1
4. Professor G. A. Kildall Department of Mathematics Naval Postgraduate School Monterey, California 93940	2
5. Professor G. L. Barksdale, Jr. Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
6. Professor W. S. Brainerd Department of Mathematics Naval Postgraduate School Monterey, California 93940	1
7. LCDR John F. Leahy III, USN Commander Submarine Force, U. S. Atlantic Fleet Code N3A Norfolk, Virginia 23511	2

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Naval Postgraduate School Monterey, California 93940		2a. REPORT SECURITY CLASSIFICATION Unclassified	
		2b. GROUP	
3. REPORT TITLE A Universal Syntax Checker			
4. DESCRIPTIVE NOTES (Type of report and, inclusive dates) Master's Thesis; June 1969			
5. AUTHOR(S) (First name, middle initial, last name) John Francis Leahy III			
6. REPORT DATE June 1969		7a. TOTAL NO. OF PAGES 65	7b. NO. OF REFS 16
8a. CONTRACT OR GRANT NO.		9a. ORIGINATOR'S REPORT NUMBER(S)	
b. PROJECT NO.			
c.		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.			
10. DISTRIBUTION STATEMENT Distribution of this document is unlimited.			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Naval Postgraduate School Monterey, California 93940	
13. ABSTRACT A universal syntax checker was constructed to be utilized with a text editor in a time-sharing environment. This syntax checker is a top-down left-right slow-back parser that will provide, when supplied the syntax of any language in the Backus-normal form, a syntax check for any string written in a language described. The procedure is capable of handling left, right, and self-embedded recursive definitions.			

14

KEY WORDS

LINK A

LINK B

LINK C

ROLE

WT

ROLE

WT

ROLE

WT

Parsing

Syntax

Recognizing

Backus-normal form

Syntax-checker

thesL3518

A universal sytanx checker.



3 2768 002 11973 7

DUDLEY KNOX LIBRARY